REPORT DOCUMENTATION PAGE

Form Approved
OMB No 0704 0188

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE 01 Jul 95	3. REPORT TYPE AND DATES COVERED Annual Technical, 01 Jul 94-30 Jun95

4. TITLE AND SUBTITLE Scalable I/O for Irregular Loosely Synchronous Problems	5. FUNDING NUMBERS G N00014-94-1-0661
6. AUTHOR(S) Anurag Acharya, compiler	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) University of Maryland Department of Computer Science College Park, Maryland 20742-3255	8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) Office of Naval Research, ONR 252 DG Ballston Tower One 800 North Quincy Street Arlington, VA 22217-5660	10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT Unlimited	12b. DISTRIBUTION CODE

DISTRIBUTION STATEMENT A
Approved for public release;
Distribution Unlimited

13. ABSTRACT (Maximum 200 words)

To achieve good I/O performance on irregular, loosely synchronous problems it is necessary to work both at the application and the system support level. The first section describes our effor at developing an efficient out-of-core parallel sparse cholesky solver as an example of an irregular, loosely synchronous application and the second section describes the Jovian parallel I/O library which provides support for collective I/O operations.

14. SUBJECT TERMS I/O performance, parallel sparse cholesky solver, parallel library	15. NUMBER OF PAGES 5
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT

Report on "Scalable I/O for Irregular Loosely Synchronous Problems"

Principal Investigator: Joel Saltz
Department of Computer Science
University of Maryland, College Park MD 20742

July 1995

To achieve good I/O performance on irregular, loosely synchronous problems, it is necessary to work both at the application and the system support level. The first section describes our effort at developing an efficient out-of-core parallel sparse cholesky solver as an example of an irregular, loosely synchronous application and the second section describes the Jovian parallel I/O library which provides support for collective I/O operations.

1 Out-of-core Parallel Sparse Cholesky Solver

Many scientific and engineering applications require the solution of very large sparse linear systems of the form $Ax = b$ where A is sparse, symmetric and positive-definite. In some cases the memory requirements exceed the capacity of even the largest parallel supercomputers. Currently the largest installed memory pool has 51.2 GBs of main memory out of which a little more than 40GBs is available for user programs. The largest sparse system (with 10% sparsity and double precision arithmetic) that can be solved with 40GBs of memory contains roughly 223K equations. The demands of some applications are far beyond that limit, structural acoustics problems being an example. Submarine structural acoustics problems can require solution of sparse linear systems with 2-3 million equations. These applications require efficient out-of-core methods which hide the disk latency by overlapping I/O with computation and which take advantage of the parallelism provided by the disk arrays available on modern parallel architectures.

We have developed an out-of-core parallel sparse Cholesky solver to address the direct solution of very large sparse systems in parallel machines with large main memory and disk capacity. The solver is targeted for distributed memory parallel architectures with local disk(s) on each compute node. In principle, this is not a requirement for the out-of-core algorithm, rather it is a requirement of the current implementation. The main features of our out-of-core sparse solver are:

- Use of asynchronous I/O to overlap computation and I/O
- Blocked factorization kernels for high CPU utilization
- Parallel execution of numerical computations and I/O

The input to the solver is an ordered and permuted matrix, which has not been symbolically factored. Because of the large disk space requirements of the sparse matrices after the fill-ins, we have developed a parallel symbolic factorization routine which distributes the matrix components to the local disks of processors. The role of the symbolic factorization routine is to compute the row indices of the lower triangular Cholesky factor, accounting fill-in entries caused by the Cholesky factorization. The row indices of Cholesky factor, as well as the row indices and the numerical values of the input matrix is stored in binary after the completion of the symbolic factorization routine.

The next component of our parallel out-of-core sparse solver is the block partitioner. This component is responsible for producing the supernodal blocks that lists the nonzeros in row-major order within a supernode. The sparse matrix is partitioned in two dimensions along the boundaries of the supernodes. The blocks are distributed among the computational nodes using the scatter decomposition. Each processor stores the blocks in its local disk along with the information that describes the block stored.

We have taken a supernodal approach in our solver. The only static data in the solver is the size and offset of each supernodes' blocks, which is proportional to the number of supernodes in size. The computation proceeds from left-to-right in a right-looking variant of sparse Cholesky. At each factorization step, all of the updates originating from a supernode is applied to the rest of the matrix to the right of that supernode. Each processor only updates the local portion of the matrix for which it is responsible for.

The unit of data transfer between disk and the memory is a whole supernode for reads, and a block for writes. The writes are block-oriented because an update to a supernode might affect only a portion of the supernode being updated. The solver keeps track of modified blocks and only the modified blocks are written back to the disk after the update. All of the disk operations use asynchronous I/O to mask the I/O latency.

The unit of communication between processors is a block. Due to the distribution scheme for the blocks, interprocessor communication for I/O is limited to a subset of processors, and the size of the subset grows proportional to the square root of the total number of processors (see previous reports). The communication protocol uses asynchronous communication primitives for sends and blocking primitives for for the receives (the implementation uses message passing for interprocessor communication).

The solver has been implemented on the 16 processor IBM SP-2 at the University of Maryland. We are currently in the process of evaluating its performance.

2 The Jovian Parallel I/O Library

The Jovian runtime library was designed to address the I/O bottleneck of parallel applications by optimizing the performance of multiprocessor architectures that include multiple disks or disk arrays. Jovian accomplishes this by presenting a *collective I/O* model to the programmer allowing the I/O for parallel programs, executing in a *loosely synchronous* manner, to be optimized. A key objective of Jovian is to aggregate disk access requests in a way that makes it possible to present each secondary storage device with a minimal number of disk access requests, thereby reducing disk latency. It uses a set of coalescing processes to aggregate the requests and to stage the I/O. These coalescing processes correspond to the I/O service nodes on most extant large cardinality multiprocessors. The library is able to analyze requests for array sections and to determine which disk the data is located on. The number of coalescing processes can be varied.

There are two complementary views for accessing an out-of-core data structure (residing on secondary storage) from each process running a parallel program. With a *global* view, access to out-of-core data requires copying a globally specified subset of an out-of-core data structure distributed across disks from or to a globally specified subset of a data structure distributed across the processes (an *in-core* data structure). In order to implement such accesses, the library needs to know the distributions of both the out-of-core data structure and the in-core data structure, and also requires a description of the requested data transfer. The in-core and out-of-core data distribution information can be defined compactly in either a single global data descriptor or multiple data descriptors.

With a *distributed* view, each process effectively requests only the part of the data structure that it requires. In that case, the application is responsible for translating local, in-core, data structure addresses into the correct global addresses for accessing the entire out-of-core data structure (perhaps through calls to a runtime library such as CHAOS or Multi-block PARTI).With a distributed view, the user program is responsible for providing the I/O system with only the requests for that process, so the burden of translating local addresses for a distributed data structure into the corresponding global, out-of-core, addresses is on the user program. The I/O system is then only responsible for optimizing the accesses to maximize bandwidth and minimize latency to secondary storage. Currently, Jovian implements the distributed view.

The performance of Jovian was evaluated using three application templates, each modeling a structured grid application. The experiments were performed on the 128 node IBM SP-1 at the Argonne National Laboratory. For each application, the out-of-core data is a two-dimensional array striped across the disks by blocks of rows (in HPF terms, with a *(block, *)* distribution). The disks used were $\sim 1GB$ SCSI with a maximum transfer rate of about 3MB/sec.

The structured grid applications templates were parallelized using the Multi-block PARTI runtime library. This library provides the functionality to lay out distributed arrays in a user specified way. For two of the application templates, the in-core structured grid was partitioned across processors in both dimensions (in HPF terms, with a *(block,block)* distribution).

The first application template involved reading an $N \times N$, (block, block)-distributed, in-core structured grid from disk (stored by blocks of rows) on 8, 16 and 32 application processes, with the number of disks ranging from 2 up to the number of application processes. The array size was varied from $4K \times 4K$ double precision floating point numbers (128 MB) up to $8K \times 8K$ (512 MB). The performance results are shown in Figures 1, 2 and 3. The measured raw disk transfer rate averaged about 2.0 MB/sec.

The second application template is similar to the first, except that it utilizes an in-core (*block,**) distribution. Thus the in-core array distribution matches the out-of-core distribution. The performance results for this application are shown in Table 1.

The variability in the disk transfer rate can be attributed to disk fragmentation on the Unix filesystem mounted on each disk. From the minimum and maximum rates, we can see that variability in the total *per* application process transfer rate is a strong function of the associated disk transfer rate. The overhead of the library, which causes the difference between the disk read rate and the effective data rate seen by the application processes shows

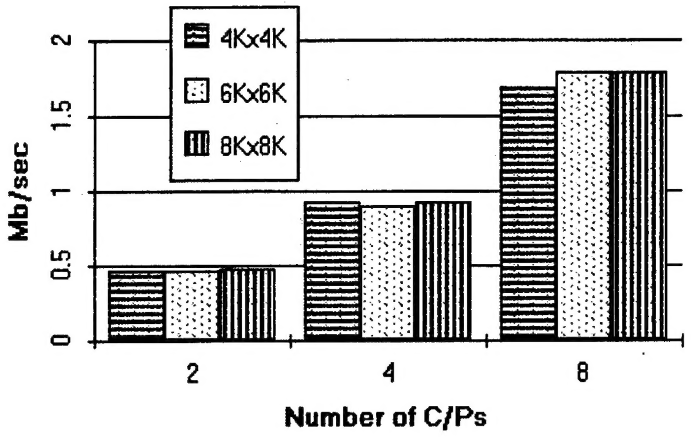

Figure 1: Non-conforming distribution, minimum effective I/O bandwidth (8 processors)

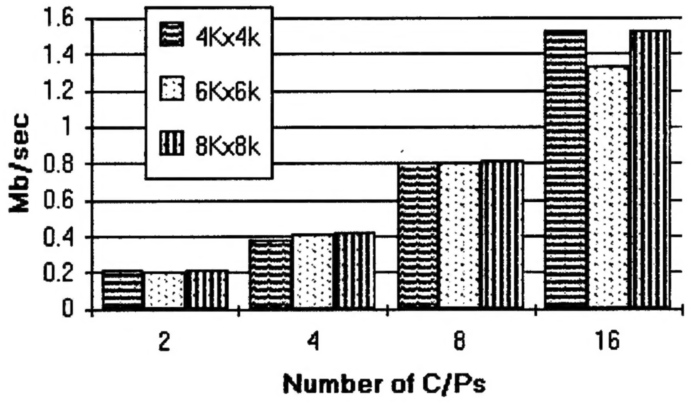

Figure 2: Non-conforming distributions, minimum effective I/O bandwidth (16 processors)

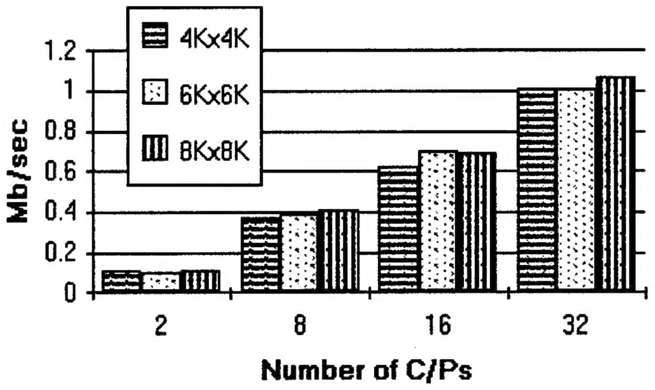

Figure 3: Non-conforming distributions, minimum effective I/O bandwidth (32 processors)

Global Grid Size	#Processors	#Disks	Fastest Disk Read		Slowest Disk Read	
			Disk Read	Jovian Read	Disk Read	Jovian Read
$4K \times 4K$	8	8	2.2	1.8	1.9	1.6
	16	16	2.2	1.8	1.6	1.4
	32	32	2.2	1.6	1.8	1.3
$6K \times 6K$	8	8	2.2	1.9	2.1	1.8
	16	16	2.2	1.8	1.8	1.5
	32	32	2.2	1.8	1.7	1.5
$8K \times 8K$	8	8	2.2	1.9	2.2	1.9
	16	16	2.2	1.9	2.1	1.8
	32	32	2.2	1.8	1.8	1.5

Table 1: Performance results for (block, *) distributed grid. The numbers in cols 4-7 are in MB/sec

Global Grid Size	A/P Write Rate	
	4 procs, 4 disks	8 procs, 8 disks
4Kx4K	0.89	0.84
6Kx6K	0.96	0.81

Table 2: A/P write for (block, block) distributed grid (MB/sec)

only small variations.

The third application template performed a collective write of a *(block, block)*-distributed structured grid. Table 2 shows the performance results. The significant difference between the read and write transfer rates is due to the fact that the library performs a *read-modify-write* on the disk blocks. This could potentially save time when there are many non-contiguous block requests remaining after coalescing (as is the case for unstructured meshes). The test application made contiguous requests, which eliminated the need for the read operation in the *read-modify-write.*